Create, Read, Update, and Delete (CRUD) DB Operations Using PDO

Make sure if PDO is enabled in your PHP.

Option 1: Use phpinfo().

```
<?php
phpinfo();
?>
```

Option 2: Use command line:

```
> php -m | grep pdo
pdo_mysql
pdo_sqlite
PDO
```

If PDO is disabled, uncomment the following line:

```
extension=pdo_mysql
```

Recommendation: Use `declare(strict_types=1);`

`declare(strict_types=1);` enables **strict type checking** in PHP:

- Forces strict type declarations for function parameters and return values
- Prevents automatic type conversion that could lead to bugs
- Makes code more predictable and safer
- Recommended for modern PHP development

Connection

```php
<?php
declare(strict_types=1);

// --- Database connection (PDO) ---
$servername = "localhost";
$username   = "root";
$password   = "123456"; // Your password
$dbname     = "studentdb";

try {
    $dsn = "mysql:host={$servername};dbname={$dbname};charset=utf8mb4";
    $options = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION, // throw exceptions
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => false,                  // use native prepares
    ];
    $pdo = new PDO($dsn, $username, $password, $options);
    echo "Connected successfully<br><br>";
} catch (PDOException $e) {
    exit("Connection failed: " . htmlspecialchars($e->getMessage()));
}
```

CREATE - Insert new student records

```php
echo "<h2>CREATE – Adding New Students</h2>";

try {
    $sql  = "INSERT INTO students (name, age, major) VALUES (:name, :age, :major)";
    $stmt = $pdo->prepare($sql);

    // 1) Alice Johnson
    $stmt->execute([
        ':name'  => "Alice Johnson",
        ':age'   => 22,
        ':major' => "Computer Science",
    ]);
    echo "Student 'Alice Johnson' added successfully<br>";

    // 2) Bob Smith
    $stmt->execute([
        ':name'  => "Bob Smith",
        ':age'   => 21,
        ':major' => "Mathematics",
    ]);
    echo "Student 'Bob Smith' added successfully<br>";

    echo "<br>";
} catch (PDOException $e) {
    echo "Error (CREATE): " . htmlspecialchars($e->getMessage()) . "<br><br>";
}
```

Two Use Cases of READ operation

We have two use cases to READ database element:

query + fetchAll

```php
$rows = $pdo->query($sql)->fetchAll();
```

- Use case: When your SQL has no user input or parameters.
- Why: Simpler, one-liner shortcut for “run and fetch.”
- Downside: If you manually interpolate variables into $sql, you risk SQL injection.

prepare + execute + fetch

```php
$sql  = "SELECT id, name, age, major FROM students WHERE id = :id";
$stmt = $pdo->prepare($sql);
$stmt->execute([':id' => $student_id]);
$row = $stmt->fetch();
```

- Use case: When you need parameters (like :id), especially user-supplied ones.
- Why: Prevents SQL injection (the database engine handles escaping/binding safely).
- Performance: Prepared statements can be compiled once and executed many times with different values.
- Flexibility: Can fetch one row (fetch()), multiple rows (fetchAll()), or stream results gradually.

READ - Display all student records

```php
echo "<h2>READ – All Students</h2>";

try {
    $sql    = "SELECT id, name, age, major FROM students";
    $rows   = $pdo->query($sql)->fetchAll();

    if ($rows && count($rows) > 0) {
        echo "Found " . count($rows) . " students:<br>";
        foreach ($rows as $row) {
            echo "ID: {$row['id']} – Name: {$row['name']} – Age: {$row['age']} – Major: {$row['major']}<br>";
        }
    } else {
        echo "No students found<br>";
    }
    echo "<br>";
} catch (PDOException $e) {
    echo "Error (READ all): " . htmlspecialchars($e->getMessage()) . "<br><br>";
}
```

`fetchAll()` - Returns **all rows** as an array of arrays

- **Use `fetchAll()` when:** You need all records at once

`fetch()` - Returns **one row** at a time as an array

- **Use `fetch()` when:** You need one specific record or want to process rows one by one

READ - Get specific student by ID

```php
echo "<h2>READ - Specific Student (ID = 1)</h2>";

try {
    $student_id = 1;
    $sql  = "SELECT id, name, age, major FROM students WHERE id = :id";
    $stmt = $pdo->prepare($sql);
    $stmt->execute([':id' => $student_id]);
    $row = $stmt->fetch();

    if ($row) {
        echo "Found student: {$row['name']} (Age: {$row['age']}, Major: {$row['major']})<br>";
    } else {
        echo "No student found with ID {$student_id}<br>";
    }
    echo "<br>";
} catch (PDOException $e) {
    echo "Error (READ one): " . htmlspecialchars($e->getMessage()) . "<br><br>";
}
```

UPDATE - Modify existing student record

```php
echo "<h2>UPDATE – Updating Student</h2>";

try {
    $update_id = 1;
    $new_name  = "Alice Johnson Updated";
    $new_age   = 23;
    $new_major = "Computer Engineering";

    $sql  = "UPDATE students
             SET name = :name, age = :age, major = :major
             WHERE id = :id";
    $stmt = $pdo->prepare($sql);
    $stmt->execute([
        ':name'  => $new_name,
        ':age'   => $new_age,
        ':major' => $new_major,
        ':id'    => $update_id,
    ]);

    if ($stmt->rowCount() > 0) {
        echo "Student with ID {$update_id} updated successfully<br>";
    } else {
        // Note: rowCount can be 0 if values are identical
        echo "No changes made (or student not found).<br>";
    }
} catch (PDOException $e) {
    echo "Error (UPDATE): " . htmlspecialchars($e->getMessage()) . "<br><br>";
}
```

`rowCount()` returns the **number of rows affected** by the last DELETE, INSERT, or UPDATE statement.

Use it to:

- Verify if an UPDATE/DELETE operation affected any rows
- Check if an INSERT operation was successful
- Provide feedback to users about operation results

Note: `rowCount()` can be 0 even for successful UPDATE if the new values are identical to existing ones.

DELETE - Remove a student record

```php
echo "<h2>DELETE – Removing Student</h2>";

try {
    $delete_id = 2;
    $sql  = "DELETE FROM students WHERE id = :id";
    $stmt = $pdo->prepare($sql);
    $stmt->execute([':id' => $delete_id]);

    if ($stmt->rowCount() > 0) {
        echo "Student with ID {$delete_id} deleted successfully<br>";
    } else {
        echo "No student found with ID {$delete_id}<br>";
    }
    echo "<br>";
} catch (PDOException $e) {
    echo "Error (DELETE): " . htmlspecialchars($e->getMessage()) . "<br><br>";
}
```

Comparison of the syntax differences

MySQLi and PDO for prepared statements.

MySQLi:

```php
$stmt = $conn->prepare("INSERT INTO students (name, age, major) VALUES (?, ?, ?)");
$stmt->bind_param("sis", $name, $age, $major);
$stmt->execute();
```

PDO:

```php
$stmt = $pdo->prepare("INSERT INTO students (name, age, major) VALUES (:name, :age, :major)");
$stmt->execute([':name' => $name, ':age' => $age, ':major' => $major]);
```

PDO advantages: Named parameters, cleaner syntax, no separate bind step required.

Using htmlspecialchars() function

We should use `htmlspecialchars()` when displaying error messages.

`htmlspecialchars()` prevents **XSS (Cross-Site Scripting) attacks** by:

- Converting special HTML characters to HTML entities
- Preventing malicious scripts from being executed in error messages
- Making error output safe for display in web browsers

Example: `<script>` becomes `&lt;script&gt;` and won't execute